UNIT 3

DIVIDE & CONQUER ALGORITHMS

Patricia Cuesta Ruiz

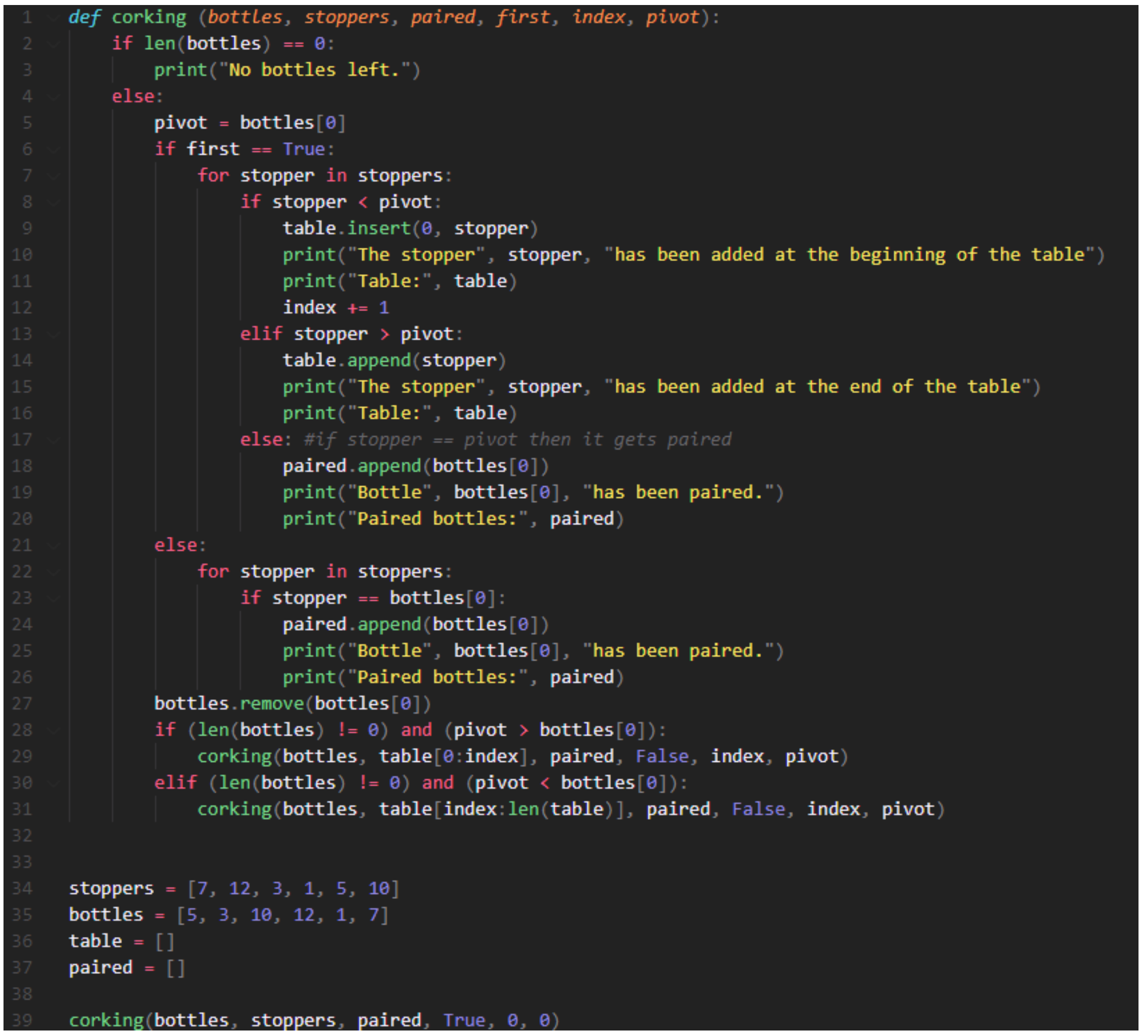
Isabel Blanco Martínez

EXERCISE 4

You want to program a robot to put cork stoppers to the bottles of a recycling factory. There are available N bottles and the N corks that cover them (N is constant in the problem), but there are a number of problems:

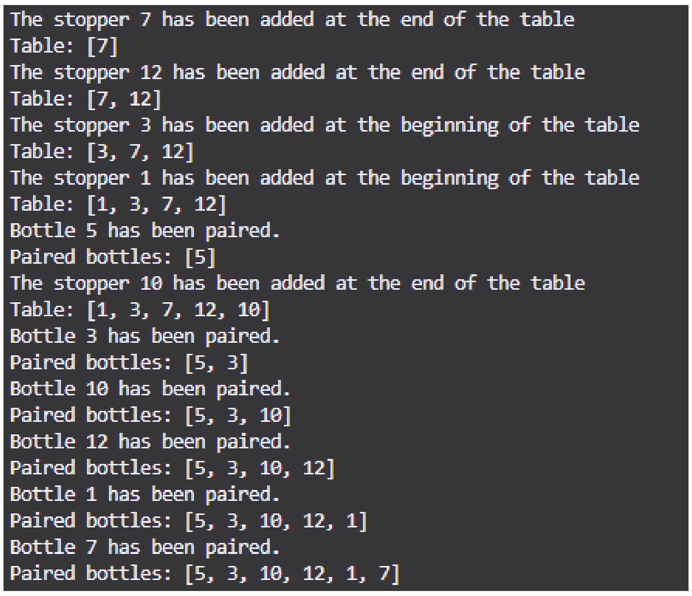
- The bottles are all different from each other, just like corks: each bottle can only be closed with a concrete cork, and each cork only serves for a specific bottle.
- The robot is prepared to close bottles, so all he knows how to do is compare corks with bottles. The robot can detect if a cork is too small, too large, or just the right size to close a bottle.
- The robot can not compare corks with each other to "sort" them by thickness, nor can it do so with the bottles.
- The robot has space available and mechanical arms to place bottles and corks at will, for example in different positions of a table, if necessary. Design the algorithm that the robot needs to plug N bottles optimally.

Code:



```python
def corking (bottles, stoppers, paired, first, index, pivot):
    if len(bottles) == 0:
        print("No bottles left.")
    else:
        pivot = bottles[0]
        if first == True:
            for stopper in stoppers:
                if stopper < pivot:
                    table.insert(0, stopper)
                    print("The stopper", stopper, "has been added at the beginning of the table")
                    print("Table:", table)
                    index += 1
                elif stopper > pivot:
                    table.append(stopper)
                    print("The stopper", stopper, "has been added at the end of the table")
                    print("Table:", table)
                else: #if stopper == pivot then it gets paired
                    paired.append(bottles[0])
                    print("Bottle", bottles[0], "has been paired.")
                    print("Paired bottles:", paired)
        else:
            for stopper in stoppers:
                if stopper == bottles[0]:
                    paired.append(bottles[0])
                    print("Bottle", bottles[0], "has been paired.")
                    print("Paired bottles:", paired)
        bottles.remove(bottles[0])
        if (len(bottles) != 0) and (pivot > bottles[0]):
            corking(bottles, table[0:index], paired, False, index, pivot)
        elif (len(bottles) != 0) and (pivot < bottles[0]):
            corking(bottles, table[index:len(table)], paired, False, index, pivot)


stoppers = [7, 12, 3, 1, 5, 10]
bottles = [5, 3, 10, 12, 1, 7]
table = []
paired = []

corking(bottles, stoppers, paired, True, 0, 0)
```

Output:



```
The stopper 7 has been added at the end of the table
Table: [7]
The stopper 12 has been added at the end of the table
Table: [7, 12]
The stopper 3 has been added at the beginning of the table
Table: [3, 7, 12]
The stopper 1 has been added at the beginning of the table
Table: [1, 3, 7, 12]
Bottle 5 has been paired.
Paired bottles: [5]
The stopper 10 has been added at the end of the table
Table: [1, 3, 7, 12, 10]
Bottle 3 has been paired.
Paired bottles: [5, 3]
Bottle 10 has been paired.
Paired bottles: [5, 3, 10]
Bottle 12 has been paired.
Paired bottles: [5, 3, 10, 12]
Bottle 1 has been paired.
Paired bottles: [5, 3, 10, 12, 1]
Bottle 7 has been paired.
Paired bottles: [5, 3, 10, 12, 1, 7]
```

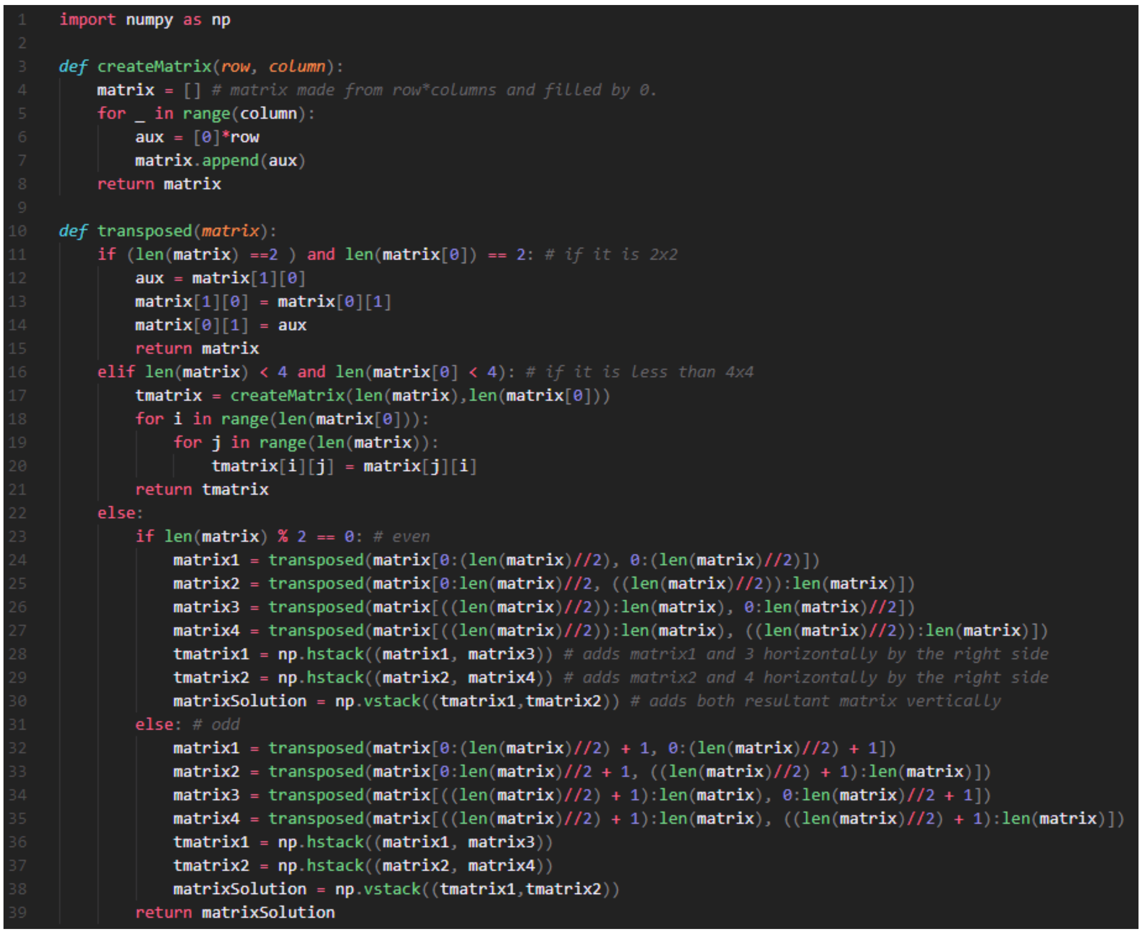
Explication:

The program is in charge of bottling bottles with the same size corks on a table. We've used a bottle as a pivot that will divide the initial list (*bottles*) into another one (which is *table*), placing the smallest ones on the left and the largest ones on the right. If the pivot finds its corresponding cork while dividing the list, it would be paired. This process will go on to the next recursive call until it finishes.

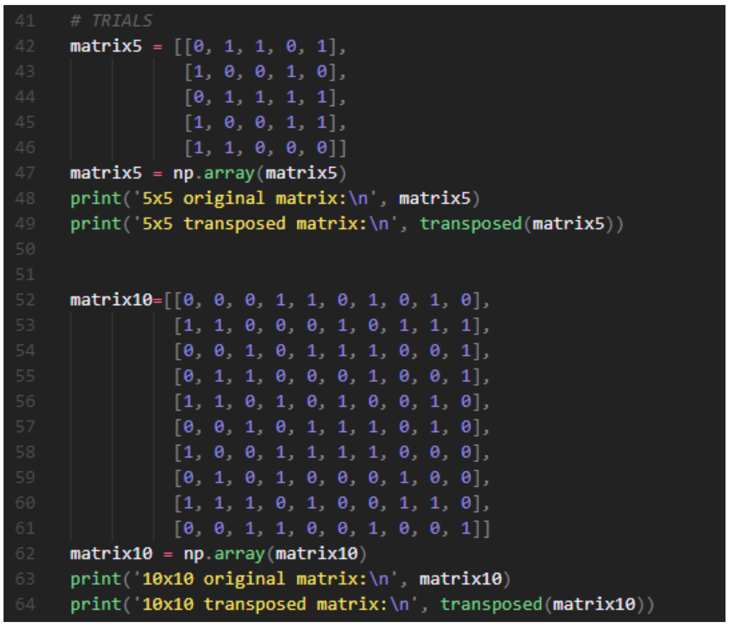
EXERCISE 7

In Abeceland, a city famous for its N beautiful squares and that maybe you know, they have a curious system of roads: from each square a street goes to all the other squares that begin with a letter that is in their name (for example, from Ace, there are streets that lead to the squares that begin with C, as Cut or Coast Squares, or by E, as East Square, but do not have streets to places like Doom, Fall or Tiara). The streets are one-way (Ace Square can go to Cut Square, but not the other way around since it does not meet the rule of the letters, obviously, other places like Ace and Cat are connected to each other in both directions). All these connections between the N squares are collected in a street map, represented by an adjacency matrix of size NxN; thus, the value of Streets[p, q] indicates whether one can go from plaza p to plaza q. April 26, festival of San Isidoro de Sevilla (patron of letters and, coincidentally, computer scientists) is approaching, and the City of Abeceland have decided to celebrate it reversing the address of all streets that connect their places. On that day you can not move from Ace to Cut, but you will be allowed to go from Cut to Ace to Aro; Obviously, Ace and Cat will remain connected to each other. To formally design a standard Divide and Conquer algorithm that, having as data the street map of the city (represented by the adjacency matrix), obtain the new street map valid for the day of San Isidoro of Seville, indicating the data structures that are used.

Code:

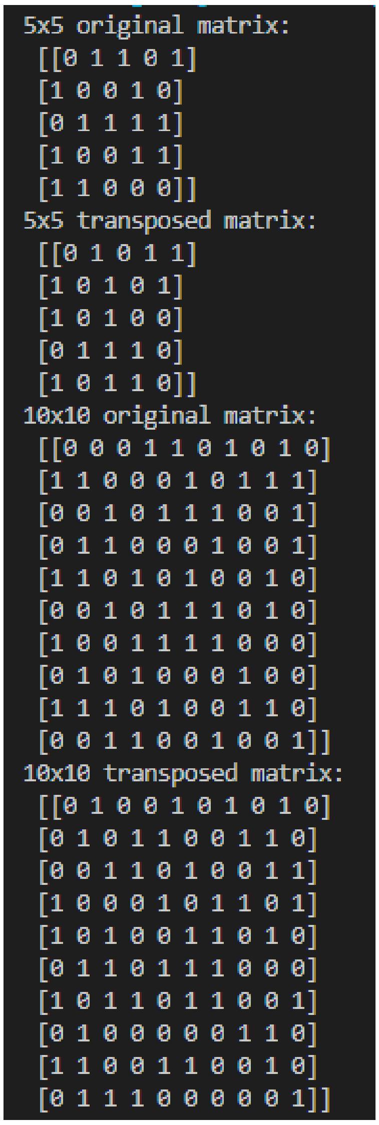


```python
import numpy as np

def createMatrix(row, column):
    matrix = [] # matrix made from row*columns and filled by 0.
    for _ in range(column):
        aux = [0]*row
        matrix.append(aux)
    return matrix

def transposed(matrix):
    if (len(matrix) ==2 ) and len(matrix[0]) == 2: # if it is 2x2
        aux = matrix[1][0]
        matrix[1][0] = matrix[0][1]
        matrix[0][1] = aux
        return matrix
    elif len(matrix) < 4 and len(matrix[0] < 4): # if it is less than 4x4
        tmatrix = createMatrix(len(matrix),len(matrix[0]))
        for i in range(len(matrix[0])):
            for j in range(len(matrix)):
                tmatrix[i][j] = matrix[j][i]
        return tmatrix
    else:
        if len(matrix) % 2 == 0: # even
            matrix1 = transposed(matrix[0:(len(matrix)//2), 0:(len(matrix)//2)])
            matrix2 = transposed(matrix[0:len(matrix)//2, ((len(matrix)//2)):len(matrix)])
            matrix3 = transposed(matrix[((len(matrix)//2)):len(matrix), 0:len(matrix)//2])
            matrix4 = transposed(matrix[((len(matrix)//2)):len(matrix), ((len(matrix)//2)):len(matrix)])
            tmatrix1 = np.hstack((matrix1, matrix3)) # adds matrix1 and 3 horizontally by the right side
            tmatrix2 = np.hstack((matrix2, matrix4)) # adds matrix2 and 4 horizontally by the right side
            matrixSolution = np.vstack((tmatrix1,tmatrix2)) # adds both resultant matrix vertically
        else: # odd
            matrix1 = transposed(matrix[0:(len(matrix)//2) + 1, 0:(len(matrix)//2) + 1])
            matrix2 = transposed(matrix[0:len(matrix)//2 + 1, ((len(matrix)//2) + 1):len(matrix)])
            matrix3 = transposed(matrix[((len(matrix)//2) + 1):len(matrix), 0:len(matrix)//2 + 1])
            matrix4 = transposed(matrix[((len(matrix)//2) + 1):len(matrix), ((len(matrix)//2) + 1):len(matrix)])
            tmatrix1 = np.hstack((matrix1, matrix3))
            tmatrix2 = np.hstack((matrix2, matrix4))
            matrixSolution = np.vstack((tmatrix1,tmatrix2))
        return matrixSolution
```



```python
# TRIALS
matrix5 = [[0, 1, 1, 0, 1],
           [1, 0, 0, 1, 0],
           [0, 1, 1, 1, 1],
           [1, 0, 0, 1, 1],
           [1, 1, 0, 0, 0]]
matrix5 = np.array(matrix5)
print('5x5 original matrix:\n', matrix5)
print('5x5 transposed matrix:\n', transposed(matrix5))


matrix10=[[0, 0, 0, 1, 1, 0, 1, 0, 1, 0],
          [1, 1, 0, 0, 0, 1, 0, 1, 1, 1],
          [0, 0, 1, 0, 1, 1, 1, 0, 0, 1],
          [0, 1, 1, 0, 0, 0, 1, 0, 0, 1],
          [1, 1, 0, 1, 0, 1, 0, 0, 1, 0],
          [0, 0, 1, 0, 1, 1, 1, 0, 1, 0],
          [1, 0, 0, 1, 1, 1, 1, 0, 0, 0],
          [0, 1, 0, 1, 0, 0, 0, 1, 0, 0],
          [1, 1, 1, 0, 1, 0, 0, 1, 1, 0],
          [0, 0, 1, 1, 0, 0, 1, 0, 0, 1]]
matrix10 = np.array(matrix10)
print('10x10 original matrix:\n', matrix10)
print('10x10 transposed matrix:\n', transposed(matrix10))
```

Output:



```
5x5 original matrix:
 [[0 1 1 0 1]
 [1 0 0 1 0]
 [0 1 1 1 1]
 [1 0 0 1 1]
 [1 1 0 0 0]]
5x5 transposed matrix:
 [[0 1 0 1 1]
 [1 0 1 0 1]
 [1 0 1 0 0]
 [0 1 1 1 0]
 [1 0 1 1 0]]
10x10 original matrix:
 [[0 0 0 1 1 0 1 0 1 0]
 [1 1 0 0 0 1 0 1 1 1]
 [0 0 1 0 1 1 1 0 0 1]
 [0 1 1 0 0 0 1 0 0 1]
 [1 1 0 1 0 1 0 0 1 0]
 [0 0 1 0 1 1 1 0 1 0]
 [1 0 0 1 1 1 1 0 0 0]
 [0 1 0 1 0 0 0 1 0 0]
 [1 1 1 0 1 0 0 1 1 0]
 [0 0 1 1 0 0 1 0 0 1]]
10x10 transposed matrix:
 [[0 1 0 0 1 0 1 0 1 0]
 [0 1 0 1 1 0 0 1 1 0]
 [0 0 1 1 0 1 0 0 1 1]
 [1 0 0 0 1 0 1 1 0 1]
 [1 0 1 0 0 1 1 0 1 0]
 [0 1 1 0 1 1 1 0 0 0]
 [1 0 1 1 0 1 1 0 0 1]
 [0 1 0 0 0 0 0 1 1 0]
 [1 1 0 0 1 1 0 0 1 0]
 [0 1 1 1 0 0 0 0 0 1]]
```

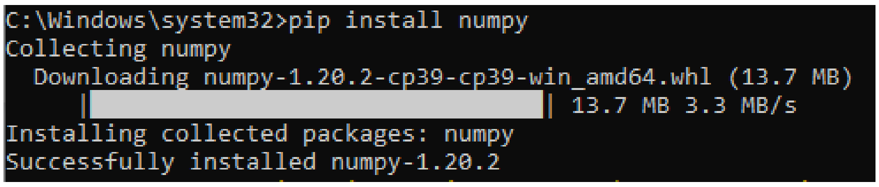
Explanation:

To solve this program, we need to obtain the transposed matrix from a given matrix. In order to do this, we will need to use the library **numpy**. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays.

To install NumPy in our computer we must follow the following steps:

1. We access through the *Control panel* to *Edit the system environment variables site*.
2. A small window pops up. We select *Environment Variables…* on the bottom side.
3. On Systems variables we look for the variable named Path. We need to add two roots:
    a. D:\Program Files\Python\
    b. D:\Program Files\Python\Scripts\

   Then we accept and Apply.
4. Then we enter the command prompt as an admin.
5. In order to install NumPy we need to write the following: *pip install numpy*
   This will be shown after the installation:



```
C:\Windows\system32>pip install numpy
Collecting numpy
  Downloading numpy-1.20.2-cp39-cp39-win_amd64.whl (13.7 MB)
     |████████████████████████████████| 13.7 MB 3.3 MB/s
Installing collected packages: numpy
Successfully installed numpy-1.20.2
```

Once we have it installed, we can begin to explain the code.

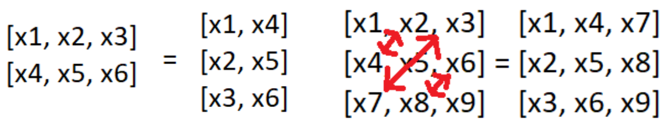
The first thing we do is to create a function which creates a matrix NxN filled by 0 used later.

Then we have the transposed algorithm, which oversees the Divide & Conquer method. We have considered 3 conditions:

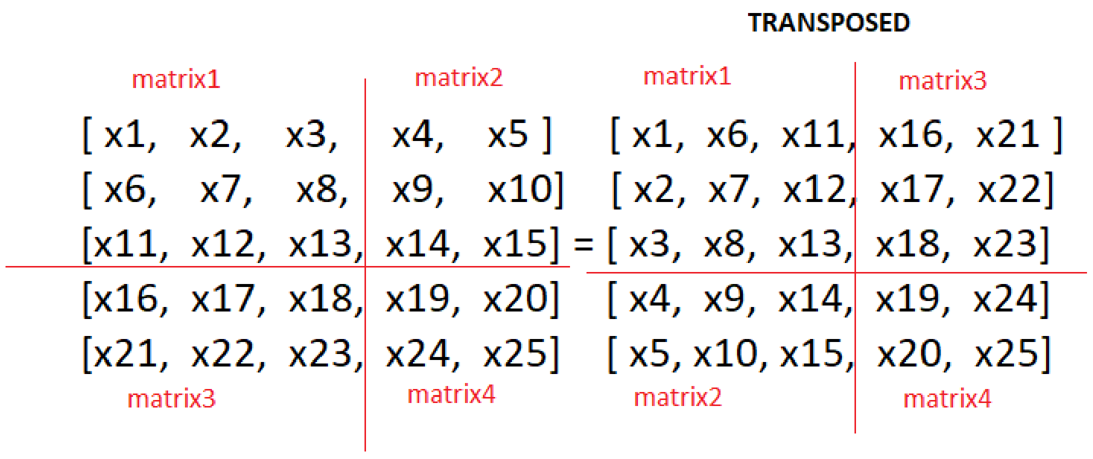
- **The matrix is 2x2**: in this case we only need to substitute two elements.

[x1, x2]
[x3, x4]

- **The matrix is less than 4x4**: we create a matrix with the needed size and place the elements where they need to be.



- **The matrix is bigger**: we will resolve each matrix individually in a recursive way, in order to divide the effort.
  In case of being even, we divide the original matrix by half the length.
  In case of being odd, we calculate the half and add 1 such as:



Once we have all the divided matrix transposed, we will use NumPy and its stacks to create two auxiliars matrixes. First, we add matrix1 and matrix 3 horizontally and name it tmatrix1. Then, we add matrix2 and matrix4 horizontally and we name it tmatrix2. What we need to do now is to join these two auxiliar matrixes, but now we need to do it in a vertical way, and we get the matrixSolution.